**BIL 206 ALGORİTMA ANALİZİ VE TASARIMI**

**DÖNEM PROJESİ RAPORU**

**PROBLEM**

Bir araç belli bir mesafeye gidecektir ve yolda kalmayacak şekilde istasyonlardan benzin alması gerekmektedir. Ancak başladıktan ve bir istasyondan benzin aldıktan sonra **k** kilometre benzin alamamaktadır ve deposundaki benzinle maksimum **f** kilometre gidebilmektedir. İstasyonların başlangıç noktasına göre mesafeleri ve bu istasyonlara ait benzin fiyatları bilinmektedir. Benzin fiyatları o istasyon için sabittir ve o istasyondan alınan benzin miktarına göre değişiklik göstermez. Bir istasyondan benzin alındıktan sonra depo tamamen dolar. En az maliyetle bu problemi 4 farklı algoritma ile çözümlenmesi istenmektedir.

**Problemdeki Değişkenler**

**mList :** İstasyonların başlangıç noktalarına göre mesafelerinin bulunduğu dizi. **pList :** İstasyonlardaki benzin fiyatlarının bulunduğu dizi.

**length :** mList ve pList dizilerini uzunluğu.

**k :** Bir istasyonda durduktan sonra durmadan devam edilmesi gereken mesafe. **f :** Depodaki benzin ile gidilebilecek maksimum mesafe.

**ending :** Varış mesafesi.

**1. Brute Force**

Brute Force algoritmasında “Tüm ihtimalleri bul ve bunlar arasından kısıtları sağlayan en düşük maliyetli olanı seç.” mantığını kullandım. Bunun için https://www.geeksforgeeks.org/print all-possible-combinations-of-r-elements-in-a-given-array-of-size-n/ adresindeki kodu kendime göre düzenleyerek küçükten büyüğe olacak şekilde tüm sıralı kombinasyonların bulunduğu double sayı vektörü tutan bir vektör oluşturdum. Bu vektörü oluştururken fiyatları tutan başka bir vektör daha oluşturdum. En sonunda ise kısıtları uygulayarak bu vektör içerisinden en düşük fiyata sahip olan vektör indexini yani “**global optimum**”u bulmuş oldum. En son ise istasyon indexleri ve fiyatları, toplam maliyet ile yazdırılıyor.

Pseudocode

**ALGORITHM** *BruteForce(mList, pList, length, k, f, ending)*

**for** i ←0 to length **do**

add all combinations to vector of vector *all* with *allComb()* function

**for** i ←0 to all.size() **do**

find the cost of each double vector in vector *all* and add to vector *prices*

**for** i ← 0 to all.size() **do**

find the lowest cost for constrains

**print** results

**FUNCTION** *allComb(array[], n, r, vector stations)*

data[r]

call function *combinationUtil()*

**FUNCTION** *combinationUtil(array[], data[], start, end, index, r, vector stations)* **if** index **equal** r **do**

add *data* to *stations*

**return**

**for** i ←*start* to (*end* **and** *end –* i+1 **greater than equal to** *r – index)* **do**

*data[i]* **equal** *i*

call *combinationUtil()*

Asimptotik olarak çalışma zamanı, dış fonksiyonlara bağlı oldudğu için biraz karışık. Tüm kombinasyonların olduğu vektörü oluşturmak **~O(2^n\*n)** zaman alıyor. Fiyatların bulunduğu vektörü oluşturmak yine **~O(2^n\*n)** zaman alıyor.

Tüm kombinasyon dizisi içerisinde dolaşarak şartları sağlayan ve maliyeti en az olanı bulmak **~O(2^n)** zaman alıyor.

Toplamda **~O(2(2^n\*n) + 2^n)** zaman olur. Kısaca **~O(2^n \* n)** zamanda çalışıyor diyebiliriz.

**2. Greedy**

Greedy algoritmasının mantığını “Mevcut depondaki benzin ile kısıtlara uygun olarak gidebildiklerin arasında değeri (değer = başladığın yerden itibaren o istasyona kadar alınan yol / o istasyondaki benzin maliyeti) en yüksek olanı seç ve sona gelene kadar tekrarla.” olarak belirledim. Bu algoritmaya göre depoda bulunan benzin ile k km durmadan ve f km’yi geçmeyecek şekilde gidilebilen istasyonlardan değeri yüksek olana, varış mesafesine ulaşana kadar gidilir. Eğer mevcut depodaki benzin ile gidilebilecek en son istasyon k km’den az mesafede ise veya varış mesafesine ulaşılamamışsa hata vererek koddan çıkar. Burada bulunan sonuç “**global optimum” değildir** çünkü o anlık mantıklı olan karar sonrası için her zaman en iyi karar olmayacağından her zaman en düşük maliyeti elde etmeyi sağlamaz ancak en düşük maliyete yakın bir maliyet bulmayı sağlar. En son ise istasyon indexleri ve fiyatları, toplam maliyet ile yazdırılıyor.

Pseudo Code

**ALGORITHM** G*reedy(mList, pList, length, k, f, ending)*

**while** not *ending* **do**

stop at most valuable station in you can go for constrains, refuel

**if** location of current station – location of last station you went < k **do**

**exit**

**print** results

Asimptotik olarak çalışma zamanı, n yerine mList ve pList’in uzunluğu olan length olarak aldığımızda ~**O(nlogn)** oluyor. Koda eklediğim

if(mList[i] > km+f)

break;

satırı eğer mevcut depoyla gidilebilecek en son istasyonda isen diğer istasyonları kontrol etmemeyi sağladığından çalışma zamanı **O(nlogn)**’den küçük ama yine de yakın oluyor. Tabii bu azalma n arttıkça daha da fazlalaşıyor.

**3. Decrease&Conquer**

Bu problem için D&C’nin temel mantığı “Greedy” algoritmasına çok benziyor ve çalışma zamanları neredeyse aynı. Temel ayırım ise D&C’nin recursive olarak çalışması. İki fonksiyon var : Ana fonksiyon ve recursive fonksiyon. Temel mantığı aynı dememin sebebi ise Greedy’de elindeki depo ile maksimum ne kadar gidilebileceklerinden değerli olana göre benzin alınırken D&C’de yeniden çağrılan recursive fonksiyona başlangıç istasyonu olarak durulan istasyonun indexi verilmesi. Yani problem sanki o istasyondan başlamış gibi. Bu recursive fonksiyonun işleme mantığı ise, eğer indexi verilen istasyon bitiş noktasına depodaki benzin ile gidilebilecek mesafede değil ise o istasyonun indexi ile fonksiyonu yeniden çağır, eğer öyle ise **return** yap. Fonksiyon **void** olduğu için bu durumda sadece “*return;”* yapıyor ve geriye doğru fonksiyondan çıkılıyor. Tüm bu istasyonlar ise ana fonksiyonda bulunan vectorlerde tutuluyor ve bu vectorlerin adresi recursive fonksiyona verilerek bu vectorler o recursive fonksiyon için sanki global olmuş oluyor. En son ise istasyon indexleri ve fiyatları, toplam maliyet ile yazdırılıyor.

Pseudocode

**ALGORITHM** *Decrease&Conquer(mList, pList, length, k, f, ending)*

call function *subDC*

**print** results

**FUNCTION** *subDC(index, vector stations, vector prices)*

**if** not *ending* **do**

find the most valuable station you can go with fuel you own for constrains

write station index and price to vector and call yourself as starting point current station **else**

**return**

Asimptotik olarak çalışma zamanı neredeyse Greedy algoritması ile aynı yani n olarak mList ve pLİst’in uzunluğu olan length’i aldığımızda ~**O(nlogn)** oluyor. Ayrıca yine kodda bulunan if(mList[i] > km+f)

break;

satırı sayesinde çalışma zamanı **O(nlogn)**’den küçük oluyor ve bu miktar n arttıkça artıyor.

**4. Dynamic Programming**

Dinamik programlamayı “Aralarında kısıtlara uygun olarak direkt bağlantı bulunan yolları tamamı 0 olarak doldurulmuş bir tabloda, iki indexin kesiştiği yere indexi büyük olanın istasyonun price’ını koy ve ardından bu tabloda 0 olmayan indexlerden küçük olanı bir vektöre diğerini başka bir vektöre ekle. Daha sonra bu yollardan sona ulaşan ve kısıtları sağlayıp en az maliyetli bir rota oluştur.” şeklinde özetleyebiliriz. Rota aralarında direkt bağlantı bulunan istasyonlardan oluştuğundan Brute Force’daki gibi rastgele her ihtimale değil, bu istasyonların ve bu istasyonlar arası maliyetin tutulduğu bir tablo sayesinde sona ulaşan ve kısıtlara uygun tüm ihtimallere bakılır ve en düşük maliyetli rota seçilir. Bu seçim işlemi başlangıcı kısıta uygun olarak verilen recursive bir fonksiyon sayesinde gerçekleşiyor. Bu fonksiyon aralarında bağlantı bulunan tüm yolları bir vektöre ekliyor. Daha sonra bu vektörden, kısıtllar tekrar kontrol edilerek en düşük maliyetli rota seçiliyor. En son ise istasyon indexleri ve fiyatları, toplam maliyet ile yazdırılıyor.

Pseudocode

**ALGORITHM** *DynamicProgramming(mList, pList, length, k, f, ending)*

find stations with direct connections for constrains and **add** them to *array* and vector *source, dest* find routes with function *findPaths()* and **add** them to vector *station*

find the lowest cost route for constrains in vector *stations*

**print** results

**FUNCTION** *findPaths(source, dest, stations, temp, index)*

**if** *ending* ***do***

**add** *temp* to *stations*

**add** *temp* to new *vector*

**for** i ← *index* to *source.size()* **do**

**if** last index of *temp* **equal** *source[i]*

*findPaths(source, dest, stations, vector, i)*

Asimptotik olarak çalışma zamanı için tam olarak şu denilemez çünkü aralarında direkt bağlantı olan yollar önceden kestirilemez. n’i mList ve pList’in uzunluğu olan length olarak alırsak ralarında bağlantı olan istasyonları bulmak **O(n^2)** zaman alıyor. Tahmini olarak aralarında bağlantı bulunan istasyonları maksimum **2n** olarak alırsak rotaları bulmak **O(2^(2n))** ve bu bulunan yollar arasındaki en az maliyetliyi kısıtlara uygun olarak bulmak en kötü durumda **O(2^(2n))** zaman alıyor. Sonuç olarak en kötü durumda **O(n^2 + 2\*2^(2n))** zaman alıyor. Kısaca toplamda**O(n^2 + 2^n)** zaman alıyor diyebiliriz.

**Algoritmaların Input Büyüklüklerine Göre Çalışma Zamanları**

n = 5 için

Brute Force : 310 mikrosaniye, 0.000310 saniye

Greedy : 47 mikrosaniye, 0.000047 saniye

Decrease&Conquer : 46 mikrosaniye, 0.000046 saniye

Dynamic Programming : 69 mikrosaniye, 0.000069 saniye

n = 10 için

Brute Force : 6311 mikrosaniye, 0.006311 saniye

Greedy : 19 mikrosaniye, 0.000019 saniye

Decrease&Conquer : 17 mikrosaniye, 0.000017 saniye

Dynamic Programming : 74 mikrosaniye, 0.000074 saniye

n = 25 için

Brute Force : 47243424 mikrosaniye, 47.243424 saniye

Greedy : 22 mikrosaniye, 0.000022 saniye

Decrease&Conquer : 15 mikrosaniye, 0.000015 saniye

Dynamic Programming : 1226257 mikrosaniye, 1.226257 saniye

n = 50 için

Brute Force : Hesaplayamadı (bad\_alloc hatası)

Greedy : 175 mikrosaniye, 0.000175 saniye

Decrease&Conquer : 134 mikrosaniye, 0.000134 saniye

Dynamic Programming : Hesaplayamadı (Süreç durduruldu)

n = 100 için

Brute Force : Hesaplayamadı (bad\_alloc hatası)

Greedy : 309 mikrosaniye, 0.000309 saniye

Decrease&Conquer : 219 mikrosaniye, 0.000219 saniye

Dynamic Programming : Hesaplayamadı (bad\_alloc hatası)

Brute Force ve Dynamic Programming her zaman aynı ve en optimal sonucu buluyor. Greedy ve Decrease&Conquer’da aynı ancak çoğunlukla Brute Force ve Dynamic Programming’den farklı ama optimal sonuca yakın bir sonuç buluyor.

Input büyüklüğü arttıkça Brute Force ve Dinamik Programlama CPU’mun (i7-9750H) hesaplama kapasitesinin ötesine geçtiğinden hesap yapamıyor. Bu yüzden her ne kadar en doğru olmasada Greedy ve D&C kullanmak daha mantıklı. Greedy ve D&C arasından ise Greedy kullanmak daha mantıklı çünkü D&C özyinelemeli bir fonksiyon kullanıyor ve bazı durumlarda sınırın dışına çıkabilir.

Gördüğüm kadarıyla bu algoritmalar arasından en uygun olanı **Greedy**’dir.